9

Source Files and Programs

Form must follow function.
– Le Corbusier

Separate compilation — linking — header files — standard library headers — the one-definition rule — linkage to non-C++ code — linkage and pointers to functions — using headers to express modularity — single-header organization — multiple-header organization — include guards — programs — advice — exercises.

9.1 Separate Compilation [file.separate]

A file is the traditional unit of storage (in a file system) and the traditional unit of compilation. There are systems that do not store, compile, and present C++ programs to the programmer as sets of files. However, the discussion here will concentrate on systems that employ the traditional use of files.

Having a complete program in one file is usually impossible. In particular, the code for the standard libraries and the operating system is typically not supplied in source form as part of a user's program. For realistically-sized applications, even having all of the user's own code in a single file is both impractical and inconvenient. The way a program is organized into files can help emphasize its logical structure, help a human reader understand the program, and help the compiler to enforce that logical structure. Where the unit of compilation is a file, all of a file must be recompiled whenever a change (however small) has been made to it or to something on which it depends. For even a moderately sized program, the amount of time spent recompiling can be significantly reduced by partitioning the program into files of suitable size.

A user presents a *source file* to the compiler. The file is then preprocessed; that is, macro processing (§7.8) is done and ***#include*** directives bring in headers (§2.4.1, §9.2.1). The result of preprocessing is called a *translation unit*. This unit is what the compiler proper works on and what the C++ language rules describe. In this book, I differentiate between source file and translation unit

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

only where necessary to distinguish what the programmer sees from what the compiler considers.

To enable separate compilation, the programmer must supply declarations providing the type information needed to analyze a translation unit in isolation from the rest of the program. The declarations in a program consisting of many separately compiled parts must be consistent in exactly the same way the declarations in a program consisting of a single source file must be. Your system will have tools to help ensure this. In particular, the linker can detect many kinds of inconsistencies. The *linker* is the program that binds together the separately compiled parts. A linker is sometimes (confusingly) called a *loader*. Linking can be done completely before a program starts to run. Alternatively, new code can be added to the program (‘‘dynamically linked’’) later.

The organization of a program into source files is commonly called the *physical structure* of a program. The physical separation of a program into separate files should be guided by the logical structure of the program. The same dependency concerns that guide the composition of programs out of namespaces guide its composition into source files. However, the logical and physical structure of a program need not be identical. For example, it can be useful to use several source files to store the functions from a single namespace, to store a collection of namespace definitions in a single file, and to scatter the definition of a namespace over several files (§8.2.4).

Here, we will first consider some technicalities relating to linking and then discuss two ways of breaking the desk calculator (§6.1, §8.2) into files.

9.2 Linkage [file.link]

Names of functions, classes, templates, variables, namespaces, enumerations, and enumerators must be used consistently across all translation units unless they are explicitly specified to be local.

It is the programmer’s task to ensure that every namespace, class, function, etc. is properly declared in every translation unit in which it appears and that all declarations referring to the same entity are consistent. For example, consider two files:

```
// file1.c:
        int x = 1;
        int f() { /* do something */ }

// file2.c:
        extern int x;
        int f();
        void g() { x = f(); }
```

The *x* and *f()* used by *g()* in *file2.c* are the ones defined in *file1.c*. The keyword *extern* indicates that the declaration of *x* in *file2.c* is (just) a declaration and not a definition (§4.9). Had *x* been initialized, *extern* would simply be ignored because a declaration with an initializer is always a definition. An object must be defined exactly once in a program. It may be declared many times, but the types must agree exactly. For example:

```
// file1.c:
        int x = 1;
        int b = 1;
        extern int c;
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
// file2.c:
    int x;                  // meaning int x = 0;
    extern double b;
    extern int c;
```

There are three errors here: *x* is defined twice, *b* is declared twice with different types, and *c* is declared twice but not defined. These kinds of errors (linkage errors) cannot be detected by a compiler that looks at only one file at a time. Most, however, are detectable by the linker. Note that a variable defined without an initializer in the global or a namespace scope is initialized by default. This is *not* the case for local variables (§4.9.5, §10.4.2) or objects created on the free store (§6.2.6). For example, the following program fragment contains two errors:

```
// file1.c:
    int x;
    int f() { return x; }

// file2.c:
    int x;
    int g() { return f(); }
```

The call of *f()* in *file2.c* is an error because *f()* has not been declared in *file2.c*. Also, the program will not link because *x* is defined twice. Note that these are not errors in C (§B.2.2).

A name that can be used in translation units different from the one in which it was defined is said to have *external linkage*. All the names in the previous examples have external linkage. A name that can be referred to only in the translation unit in which it is defined is said to have *internal linkage*.

An *inline* function (§7.1.1, §10.2.9) must be defined – by identical definitions (§9.2.3) – in every translation unit in which it is used. Consequently, the following example isn't just bad taste; it is illegal:

```
// file1.c:
    inline int f(int i) { return i; }

// file2.c:
    inline int f(int i) { return i+1; }
```

Unfortunately, this error is hard for an implementation to catch, and the following – otherwise perfectly logical – combination of external linkage and inlining is banned to make life simpler for compiler writers:

```
// file1.c:
    extern inline int g(int i);
    int h(int i) { return g(i); }    // error: g() undefined in this translation unit

// file2.c:
    extern inline int g(int i) { return i+1; }
```

By default, *const*s (§5.4) and *typedef*s (§4.9.7) have internal linkage. Consequently, this example is legal (although potentially confusing):

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
// file1.c:
        typedef int T;
        const int x = 7;

// file2.c:
        typedef void T;
        const int x = 8;
```

Global variables that are local to a single compilation unit are a common source of confusion and are best avoided. To ensure consistency, you should usually place global *const*s and *inline*s in header files only (§9.2.1).

A *const* can be given external linkage by an explicit declaration:

```
// file1.c:
        extern const int a = 77;

// file2.c:
        extern const int a;

        void g()
        {
                cout << a << '\n';
        }
```

Here, *g()* will print *77*.

An unnamed namespace (§8.2.5) can be used to make names local to a compilation unit. The effect of an unnamed namespace is very similar to that of internal linkage. For example:

```
// file 1.c:
        namespace {
                class X { /* ... */ };
                void f();
                int i;
                // ...
        }

// file2.c:
        class X { /* ... */ };
        void f();
        int i;
        // ...
```

The function *f()* in *file1.c* is not the same function as the *f()* in *file2.c*. Having a name local to a translation unit and also using that same name elsewhere for an entity with external linkage is asking for trouble.

In C and older C++ programs, the keyword *static* is (confusingly) used to mean ‘‘use internal linkage’’ (§B.2.3). Don’t use *static* except inside functions (§7.1.2) and classes (§10.2.4).

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

9.2.1 Header Files [file.header]

The types in all declarations of the same object, function, class, etc., must be consistent. Consequently, the source code submitted to the compiler and later linked together must be consistent. One imperfect but simple method of achieving consistency for declarations in different translation units is to *#include header files* containing interface information in source files containing executable code and/or data definitions.

The *#include* mechanism is a text manipulation facility for gathering source program fragments together into a single unit (file) for compilation. The directive

```
#include "to_be_included"
```

replaces the line in which the *#include* appears with the contents of the file *to_be_included*. The content should be C++ source text because the compiler will proceed to read it.

To include standard library headers, use the angle brackets < and > around the name instead of quotes. For example:

```
#include <iostream>        // from standard include directory
#include "myheader.h"      // from current directory
```

Unfortunately, spaces are significant within the < > or " " of an include directive:

```
#include <  iostream  >     // will not find <iostream>
```

It may seem extravagant to recompile a file each time it is included somewhere, but the included files typically contain only declarations and not code needing extensive analysis by the compiler. Furthermore, most modern C++ implementations provide some form of precompiling of header files to minimize the work needed to handle repeated compilation of the same header.

As a rule of thumb, a header may contain:

Named namespaces	`namespace N { /* ... */ }`
Type definitions	`struct Point { int x, y; };`
Template declarations	`template<class T> class Z;`
Template definitions	`template<class T> class V { /* ... */ };`
Function declarations	`extern int strlen(const char*);`
Inline function definitions	`inline char get(char* p) { return *p++; }`
Data declarations	`extern int a;`
Constant definitions	`const float pi = 3.141593;`
Enumerations	`enum Light { red, yellow, green };`
Name declarations	`class Matrix;`
Include directives	`#include <algorithm>`
Macro definitions	`#define VERSION 12`
Conditional compilation directives	`#ifdef __cplusplus`
Comments	`/* check for end of file */`

This rule of thumb for what may be placed in a header is not a language requirement. It is simply a reasonable way of using the *#include* mechanism to express the physical structure of a program. Conversely, a header should never contain:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

Ordinary function definitions	`char get(char* p) { return *p++; }`
Data definitions	`int a;`
Aggregate definitions	`short tbl[] = { 1, 2, 3 };`
Unnamed namespaces	`namespace { /* ... */ }`
Exported template definitions	`export template<class T> f(T t) { /* ... */ }`

Header files are conventionally suffixed by *.h*, and files containing function or data definitions are suffixed by *.c*. They are therefore often referred to as ‘‘.h files’’ and ‘‘.c files,’’ respectively. Other conventions, such as *.C*, *.cxx*, *.cpp*, and *.cc*, are also found. The manual for your compiler will be quite specific about this issue.

The reason for recommending that the definition of simple constants, but not the definition of aggregates, be placed in header files is that it is hard for implementations to avoid replication of aggregates presented in several translation units. Furthermore, the simple cases are far more common and therefore more important for generating good code.

It is wise not to be too clever about the use of *#include*. My recommendation is to *#include* only complete declarations and definitions and to do so only in the global scope, in linkage specification blocks, and in namespace definitions when converting old code (§9.2.2). As usual, it is wise to avoid macro magic. One of my least favorite activities is tracking down an error caused by a name being macro-substituted into something completely different by a macro defined in an indirectly *#include*d header that I have never even heard of.

9.2.2 Standard Library Headers [file.std.header]

The facilities of the standard library are presented through a set of standard headers (§16.1.2). No suffix is needed for standard library headers; they are known to be headers because they are included using the *#include<...>* syntax rather than *#include"..."*. The absence of a *.h* suffix does not imply anything about how the header is stored. A header such as *<map>* may be stored as a text file called *map.h* in a standard directory. On the other hand, standard headers are not required to be stored in a conventional manner. An implementation is allowed to take advantage of knowledge of the standard library definition to optimize the standard library implementation and the way standard headers are handled. For example, an implementation might have knowledge of the standard math library (§22.3) built in and treat *#include<cmath>* as a switch that makes the standard math functions available without reading any file.

For each C standard-library header *<X.h>*, there is a corresponding standard C++ header *<cX>*. For example, *#include<cstdio>* provides what *#include<stdio.h>* does. A typical *stdio.h* will look something like this:

```
#ifdef __cplusplus          // for C++ compliers only (§9.2.4)
namespace std {             // the standard library is defined in namespace std (§8.2.9)

extern "C" {                // stdio functions have C linkage (§9.2.4)
#endif

        // ...
        int printf(const char* ...);
        // ...
#ifdef __cplusplus
}
}
using namespace std;        // make stdio available in global namespace
#endif
```

That is, the actual declarations are (most likely) shared, but linkage and namespace issues must be addressed to allow C and C++ to share a header.

9.2.3 The One-Definition Rule [file.odr]

A given class, enumeration, and template, etc., must be defined exactly once in a program.

From a practical point of view, this means that there must be exactly one definition of, say, a class residing in a single file somewhere. Unfortunately, the language rule cannot be that simple. For example, the definition of a class may be composed through macro expansion (ugh!), while a definition of a class may be textually included in two source files by *#include* directives (§9.2.1). Worse, a ‘‘file’’ isn’t a concept that is part of the C and C++ language definitions; there exist implementations that do not store programs in source files.

Consequently, the rule in the standard that says that there must be a unique definition of a class, template, etc., is phrased in a somewhat more complicated and subtle manner. This rule is commonly referred to as ‘‘the one-definition rule,’’ the ODR. That is, two definitions of a class, template, or inline function are accepted as examples of the same unique definition if and only if

[1] they appear in different translation units, and
[2] they are token-for-token identical, and
[3] the meanings of those tokens are the same in both translation units.

For example:

```
// file1.c:
    struct S { int a; char b; };
    void f(S*);

// file2.c:
    struct S { int a; char b; };
    void f(S* p) { /* ... */ }
```

The ODR says that this example is valid and that *S* refers to the same class in both source files. However, it is unwise to write out a definition twice like that. Someone maintaining *file2.c* will naturally assume that the definition of *S* in *file2.c* is the only definition of *S* and so feel free to change it. This could introduce a hard-to-detect error.

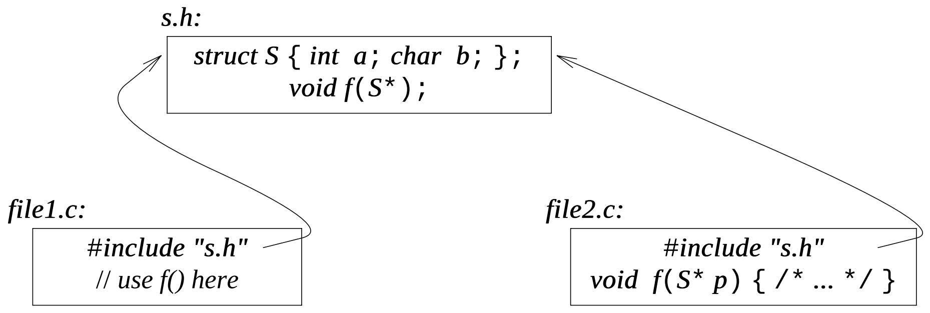
The intent of the ODR is to allow inclusion of a class definition in different translation units from a common source file. For example:

```
// file s.h:
    struct S { int a; char b; };
    void f(S*);
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
// file1.c:
    #include "s.h"
    // use f() here

// file2.c:
    #include "s.h"
    void f(S* p) { /* ... */ }
```

or graphically:



Here are examples of the three ways of violating the ODR:

```
// file1.c:
    struct S1 { int a; char b; };

    struct S1 { int a; char b; };    // error: double definition
```

This is an error because a ***struct*** may not be defined twice in a single translation unit.

```
// file1.c:
    struct S2 { int a; char b; };

// file2.c:
    struct S2 { int a; char bb; }; // error
```

This is an error because *S2* is used to name classes that differ in a member name.

```
// file1.c:
    typedef int X;
    struct S3 { X a; char b; };

// file2.c:
    typedef char X;
    struct S3 { X a; char b; };    // error
```

Here the two definitions of *S3* are token-for-token identical, but the example is an error because the meaning of the name ***X*** has sneakily been made to differ in the two files.

Checking against inconsistent class definitions in separate translation units is beyond the ability of most C++ implementations. Consequently, declarations that violate the ODR can be a source of subtle errors. Unfortunately, the technique of placing shared definitions in headers and ***#including*** them doesn't protect against this last form of ODR violation. Local typedefs and macros can change the meaning of *#include*d declarations:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
// file s.h:
      struct S { Point a; char b; };

// file1.c:
      #define Point int
      #include "s.h"
      // ...

// file2.c:
      class Point { /* ... */ };
      #include "s.h"
      // ...
```

The best defense against this kind of hackery is to make headers as self-contained as possible. For example, if class *Point* had been declared in the *s.h* header the error would have been detected.

A template definition can be *#include*d in several translation units as long as the ODR is adhered to. In addition, an exported template can be used given only a declaration:

```
// file1.c:
      export template<class T> T twice(T t) { return t+t; }

// file2.c:
      template<class T> T twice(T t);          // declaration
      int g(int i) { return twice(i); }
```

The keyword *export* means ‘‘accessible from another translation unit’’ (§13.7).

9.2.4 Linkage to Non-C++ Code [file.c]

Typically, a C++ program contains parts written in other languages. Similarly, it is common for C++ code fragments to be used as parts of programs written mainly in some other language. Cooperation can be difficult between program fragments written in different languages and even between fragments written in the same language but compiled with different compilers. For example, different languages and different implementations of the same language may differ in their use of machine registers to hold arguments, the layout of arguments put on a stack, the layout of built-in types such as strings and integers, the form of names passed by the compiler to the linker, and the amount of type checking required from the linker. To help, one can specify a *linkage* convention to be used in an *extern* declaration. For example, this declares the C and C++ standard library function *strcpy()* and specifies that it should be linked according to the C linkage conventions:

```
extern "C" char* strcpy(char*, const char*);
```

The effect of this declaration differs from the effect of the ‘‘plain’’ declaration

```
extern char* strcpy(char*, const char*);
```

only in the linkage convention used for calling *strcpy()*.

The *extern "C"* directive is particularly useful because of the close relationship between C and C++. Note that the *C* in *extern "C"* names a linkage convention and not a language. Often, *extern "C"* is used to link to Fortran and assembler routines that happen to conform to the conventions of a C implementation.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

An ***extern "C"*** directive specifies the linkage convention (only) and does not affect the semantics of calls to the function. In particular, a function declared ***extern "C"*** still obeys the C++ type checking and argument conversion rules and not the weaker C rules. For example:

```
extern "C" int f();

int g()
{
      return f(1);     // error: no argument expected
}
```

Adding ***extern "C"*** to a lot of declarations can be a nuisance. Consequently, there is a mechanism to specify linkage to a group of declarations. For example:

```
extern "C" {
      char* strcpy(char*, const char*);
      int strcmp(const char*, const char*);
      int strlen(const char*);
      // ...
}
```

This construct, commonly called a *linkage block*, can be used to enclose a complete C header to make a header suitable for C++ use. For example:

```
extern "C" {
#include <string.h>
}
```

This technique is commonly used to produce a C++ header from a C header. Alternatively, conditional compilation (§7.8.1) can be used to create a common C and C++ header:

```
#ifdef __cplusplus
extern "C" {
#endif

      char* strcpy(char*, const char*);
      int strcmp(const char*, const char*);
      int strlen(const char*);
      // ...

#ifdef __cplusplus
}
#endif
```

The predefined macro name ***__cplusplus*** is used to ensure that the C++ constructs are edited out when the file is used as a C header.

Any declaration can appear within a linkage block:

```
extern "C" {          // any declaration here, for example:
      int g1;         // definition
      extern int g2;  // declaration, not definition
}
```

In particular, the scope and storage class of variables are not affected, so ***g1*** is still a global variable

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

– and is still defined rather than just declared. To declare but not define a variable, you must apply the keyword *extern* directly in the declaration. For example:

```
extern "C" int g3;                // declaration, not definition
```

This looks odd at first glance. However, it is a simple consequence of keeping the meaning unchanged when adding *"C"* to an extern declaration and the meaning of a file unchanged when enclosing it in a linkage block.

A name with C linkage can be declared in a namespace. The namespace will affect the way the name is accessed in the C++ program, but not the way a linker sees it. The *printf()* from *std* is a typical example:

```
#include<cstdio>

void f()
{
      std::printf("Hello, ");    // ok
      printf("world!\n");        // error: no global printf()
}
```

Even when called *std::printf*, it is still the same old C *printf()* (§21.8).

Note that this allows us to include libraries with C linkage into a namespace of our choice rather than polluting the global namespace. Unfortunately, the same flexibility is not available to us for headers defining functions with C++ linkage in the global namespace. The reason is that linkage of C++ entities must take namespaces into account so that the object files generated will reflect the use or lack of use of namespaces.

9.2.5 Linkage and Pointers to Functions [file.ptof]

When mixing C and C++ code fragments in one program, we sometimes want to pass pointers to functions defined in one language to functions defined in the other. If the two implementations of the two languages share linkage conventions and function-call mechanisms, such passing of pointers to functions is trivial. However, such commonality cannot in general be assumed, so care must be taken to ensure that a function is called the way it expects to be called.

When linkage is specified for a declaration, the specified linkage applies to all function types, function names, and variable names introduced by the declaration(s). This makes all kinds of strange – and occasionally essential – combinations of linkage possible. For example:

```
typedef int (*FT)(const void*, const void*);                        // FT has C++ linkage

extern "C" {
      typedef int (*CFT)(const void*, const void*);                 // CFT has C linkage
      void qsort(void* p, size_t n, size_t sz, CFT cmp);           // cmp has C linkage
}

void isort(void* p, size_t n, size_t sz, FT cmp);                   // cmp has C++ linkage
void xsort(void* p, size_t n, size_t sz, CFT cmp);                  // cmp has C linkage
extern "C" void ysort(void* p, size_t n, size_t sz, FT cmp);        // cmp has C++ linkage

int compare(const void*, const void*);                              // compare() has C++ linkage
extern "C" int ccmp(const void*, const void*);                      // ccmp() has C linkage
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
void f(char* v, int sz)
{
        qsort(v,sz,1,&compare); // error
        qsort(v,sz,1,&ccmp);    // ok

        isort(v,sz,1,&compare); // ok
        isort(v,sz,1,&ccmp);    // error
}
```

An implementation in which C and C++ use the same calling conventions might accept the cases marked *error* as a language extension.

9.3 Using Header Files [file.using]

To illustrate the use of headers, I present a few alternative ways of expressing the physical structure of the calculator program (§6.1, §8.2).

9.3.1 Single Header File [file.single]

The simplest solution to the problem of partitioning a program into several files is to put the definitions in a suitable number of *.c* files and to declare the types needed for them to communicate in a single *.h* file that each *.c* file *#include*s. For the calculator program, we might use five *.c* files – *lexer.c*, *parser.c*, *table.c*, *error.c*, and *main.c* – to hold function and data definitions, plus the header *dc.h* to hold the declarations of every name used in more than one *.c* file.

The header *dc.h* would look like this:

```
// dc.h:

namespace Error {
        struct Zero_divide { };

        struct Syntax_error {
                const char* p;
                Syntax_error(const char* q) { p = q; }
        };
}

#include <string>

namespace Lexer {

        enum Token_value {
                NAME,           NUMBER,         END,
                PLUS='+',       MINUS='-',      MUL='*',        DIV='/',
                PRINT=';',      ASSIGN='=',     LP='(',         RP=')'
        };

        extern Token_value curr_tok;
        extern double number_value;
        extern std::string string_value;
        Token_value get_token();
}

namespace Parser {
        double prim(bool get);     // handle primaries
        double term(bool get);     // multiply and divide
        double expr(bool get);     // add and subtract

        using Lexer::get_token;
        using Lexer::curr_tok;
}

#include <map>

extern std::map<std::string,double> table;

namespace Driver {
        extern int no_of_errors;
        extern std::istream* input;
        void skip();
}
```

The keyword *extern* is used for every declaration of a variable to ensure that multiple definitions do not occur as we *#include dc.h* in the various *.c* files. The corresponding definitions are found in the appropriate *.c* files.

Leaving out the actual code, *lexer.c* will look something like this:

```
// lexer.c:

#include "dc.h"
#include <iostream>
#include <cctype>

Lexer::Token_value Lexer::curr_tok;
double Lexer::number_value;
std::string Lexer::string_value;

Lexer::Token_value Lexer::get_token() { /* ... */ }
```

Using headers in this manner ensures that every declaration in a header will at some point be included in the file containing its definition. For example, when compiling *lexer.c* the compiler will be presented with:

```
namespace Lexer {    // from dc.h
        // ...
        Token_value get_token();
}

// ...

Lexer::Token_value Lexer::get_token() { /* ... */ }
```

This ensures that the compiler will detect any inconsistencies in the types specified for a name. For example, had *get_token()* been declared to return a *Token_value*, but defined to return an *int*, the compilation of *lexer.c* would have failed with a type-mismatch error. If a definition is missing,

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

the linker will catch the problem. If a declaration is missing, some *.c* file will fail to compile.

File *parser.c* will look like this:

```
// parser.c:

#include "dc.h"

double Parser::prim(bool get) { /* ... */ }
double Parser::term(bool get) { /* ... */ }
double Parser::expr(bool get) { /* ... */ }
```

File *table.c* will look like this:

```
// table.c:

#include "dc.h"

std::map<std::string,double> table;
```

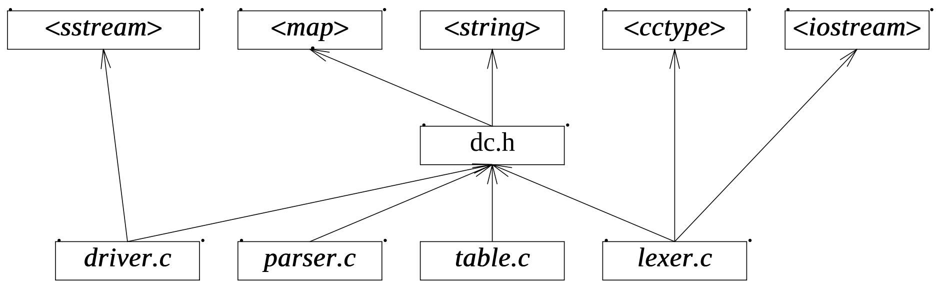
The symbol table is simply a variable of the standard library *map* type. This defines *table* to be global. In a realistically-sized program, this kind of minor pollution of the global namespace builds up and eventually causes problems. I left this sloppiness here simply to get an opportunity to warn against it.

Finally, file *main.c* will look like this:

```
// main.c:

#include "dc.h"
#include <sstream>

int Driver::no_of_errors = 0;
std::istream* Driver::input = 0;

void Driver::skip() { /* ... */ }

int main(int argc, char* argv[]) { /* ... */ }
```

To be recognized as *the* *main()* of the program, *main()* must be a global function, so no namespace is used here.

The physical structure of the system can be presented like this:



Note that the headers on the top are all headers for standard library facilities. For many forms of program analysis, these libraries can be ignored because they are well known and stable. For tiny

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

programs, the structure can be simplified by moving all *#include* directives to the common header.

This single-header style of physical partitioning is most useful when the program is small and its parts are not intended to be used separately. Note that when namespaces are used, the logical structure of the program is still represented within *dc.h*. If namespaces are not used, the structure is obscured, although comments can be a help.

For larger programs, the single header file approach is unworkable in a conventional file-based development environment. A change to the common header forces recompilation of the whole program, and updates of that single header by several programmers are error-prone. Unless strong emphasis is placed on programming styles relying heavily on namespaces and classes, the logical structure deteriorates as the program grows.

9.3.2 Multiple Header Files [file.multi]

An alternative physical organization lets each logical module have its own header defining the facilities it provides. Each *.c* file then has a corresponding *.h* file specifying what it provides (its interface). Each *.c* file includes its own *.h* file and usually also other *.h* files that specify what it needs from other modules in order to implement the services advertised in the interface. This physical organization corresponds to the logical organization of a module. The interface for users is put into its *.h* file, the interface for implementers is put into a file suffixed *_impl.h*, and the module's definitions of functions, variables, etc. are placed in *.c* files. In this way, the parser is represented by three files. The parser's user interface is provided by *parser.h*:

```
// parser.h:

namespace Parser {          // interface for users
    double expr(bool get);
}
```

The shared environment for the functions implementing the parser is presented by *parser_impl.h*:

```
// parser_impl.h:

#include "parser.h"
#include "error.h"
#include "lexer.h"

namespace Parser {          // interface for implementers
    double prim(bool get);
    double term(bool get);
    double expr(bool get);

    using Lexer::get_token;
    using Lexer::curr_tok;
}
```

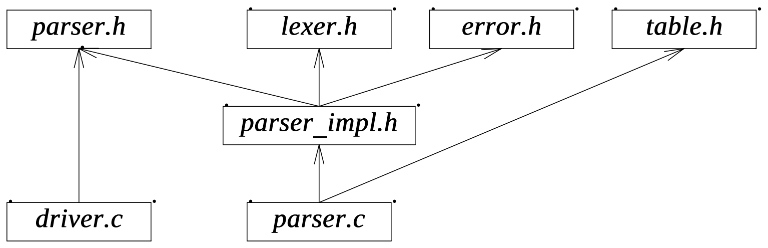
The user's header *parser.h* is *#include*d to give the compiler a chance to check consistency (§9.3.1).

The functions implementing the parser are stored in *parser.c* together with *#include* directives for the headers that the *Parser* functions need:

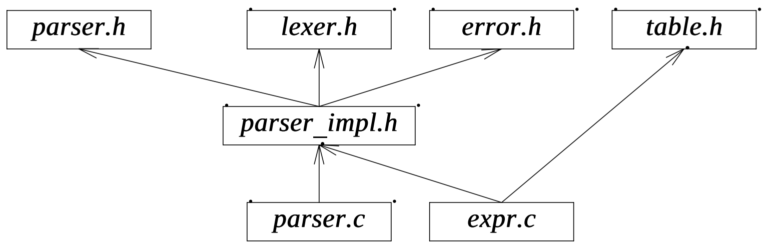
The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
// parser.c:

#include "parser_impl.h"
#include "table.h"

double Parser::prim(bool get) { /* ... */ }
double Parser::term(bool get) { /* ... */ }
double Parser::expr(bool get) { /* ... */ }
```

Graphically, the parser and the driver's use of it look like this:



As intended, this is a rather close match to the logical structure described in §8.3.3. To simplify this structure, we could have *#include*d *table.h* in *parser_impl.h* rather than in *parser.c*. However, *table.h* is an example of something that is not necessary to express the shared context of the parser functions; it is needed only by their implementation. In fact, it is used by just one function, *expr()*, so if we were really keen on minimizing dependencies we could place *expr()* in its own *.c* file and *#include table.h* there only:



Such elaboration is not appropriate except for larger modules. For realistically-sized modules, it is common to *#include* extra files where needed for individual functions. Furthermore, it is not uncommon to have more than one *_impl.h*, since different subsets of the module's functions need different shared contexts.

Please note that the *_impl.h* notation is not a standard or even a common convention; it is simply the way I like to name things.

Why bother with this more complicated scheme of multiple header files? It clearly requires far less thought simply to throw every declaration into a single header, as was done for *dc.h*.

The multiple-header organization scales to modules several magnitudes larger than our toy parser and to programs several magnitudes larger than our calculator. The fundamental reason for using this type of organization is that it provides a better localization of concerns. When analyzing

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

and modifying a large program, it is essential for a programmer to focus on a relatively small chunk of code. The multiple-header organization makes it easy to determine exactly what the parser code depends on and to ignore the rest of the program. The single-header approach forces us to look at every declaration used by any module and decide if it is relevant. The simple fact is that maintenance of code is invariably done with incomplete information and from a local perspective. The multiple-header organization allows us to work successfully ‘‘from the inside out’’ with only a local perspective. The single-header approach – like every other organization centered around a global repository of information – requires a top-down approach and will forever leave us wondering exactly what depends on what.

The better localization leads to less information needed to compile a module, and thus to faster compiles. The effect can be dramatic. I have seen compile times drop by a factor of ten as the result of a simple dependency analysis leading to a better use of headers.

9.3.2.1 Other Calculator Modules [file.multi.etc]

The remaining calculator modules can be organized similarly to the parser. However, those modules are so small that they don't require their own *_impl.h* files. Such files are needed only where a logical module consists of many functions that need a shared context.

The error handler was reduced to the set of exception types so that no *error.c* was needed:

```
// error.h:

namespace Error {
      struct Zero_divide { };

      struct Syntax_error {
            const char* p;
            Syntax_error(const char* q) { p = q; }
      };
}
```

The lexer provides a rather large and messy interface:

```
// lexer.h:

#include <string>

namespace Lexer {

      enum Token_value {
            NAME,         NUMBER,       END,
            PLUS='+',     MINUS='-',    MUL='*',      DIV='/',
            PRINT=';',    ASSIGN='=',   LP='(',       RP=')'
      };

      extern Token_value curr_tok;
      extern double number_value;
      extern std::string string_value;

      Token_value get_token();
}
```

In addition to *lexer.h*, the implementation of the lexer depends on *error.h*, *<iostream>*, and the functions determining the kinds of characters declared in *<cctype>*:

```
// lexer.c:

#include "lexer.h"
#include "error.h"
#include <iostream>
#include <cctype>

Lexer::Token_value Lexer::curr_tok;
double Lexer::number_value;
std::string Lexer::string_value;

Lexer::Token_value Lexer::get_token() { /* ... */ }
```

We could have factored out the *#include* statements for *error.h* as the *Lexer*'s *_impl.h* file. However, I considered that excessive for this tiny program.

As usual, we *#include* the interface offered by the module – in this case, *lexer.h* – in the module's implementation to give the compiler a chance to check consistency.

The symbol table is essentially self-contained, although the standard library header *<map>* could drag in all kinds of interesting stuff to implement an efficient *map* template class:

```
// table.h:

#include <map>
#include <string>

extern std::map<std::string,double> table;
```

Because we assume that every header may be *#include*d in several *.c* files, we must separate the declaration of *table* from its definition, even though the difference between *table.c* and *table.h* is the single keyword *extern*:

```
// table.c:

#include "table.h"

std::map<std::string,double> table;
```

Basically, the driver depends on everything:

```
// main.c:

#include "parser.h"
#include "lexer.h"
#include "error.h"
#include "table.h"

namespace Driver {
        int no_of_errors;
        std::istream* input;
        void skip();
}
```

```
#include <sstream>

int main(int argc, char* argv[]) { /* ... */ }
```

Because the *Driver* namespace is used exclusively by *main*( ), I placed it in *main.c*. Alternatively, I could have factored it out as *driver.h* and *#include*d it.

For a larger system, it is usually worthwhile organizing things so that the driver has fewer direct dependencies. Often, is it also worth minimizing what is done in *main*( ) by having *main*( ) call a driver function placed in a separate source file. This is particularly important for code intended to be used as a library. Then, we cannot rely on code in *main*( ) and must be prepared to be called from a variety of functions (§9.6[8]).

9.3.2.2 Use of Headers [file.multi.use]

The number of headers to use for a program is a function of many factors. Many of these factors have more to do with the way files are handled on your system than with C++. For example, if your editor does not have facilities for looking at several files at the same time, then using many headers becomes less attractive. Similarly, if opening and reading 20 files of 50 lines each is noticeably more time-consuming than reading a single file of 1000 lines, you might think twice before using the multiple-header style for a small project.

A word of caution: a dozen headers plus the standard headers for the program's execution environment (which can often be counted in the hundreds) are usually manageable. However, if you partition the declarations of a large program into the logically minimal-sized headers (putting each structure declaration in its own file, etc.), you can easily get an unmanageable mess of hundreds of files even for minor projects. I find that excessive.

For large projects, multiple headers are unavoidable. In such projects, hundreds of files (not counting standard headers) are the norm. The real confusion starts when they start to be counted in the thousands. At that scale, the basic techniques discussed here still apply, but their management becomes a Herculean task. Remember that for realistically-sized programs, the single-header style is not an option. Such programs will have multiple headers. The choice between the two styles of organization occurs (repeatedly) for the parts that make up the program.

The single-header style and the multiple-header style are not really alternatives to each other. They are complementary techniques that must be considered whenever a significant module is designed and must be reconsidered as a system evolves. It's crucial to remember that one interface doesn't serve all equally well. It is usually worthwhile to distinguish between the implementers' interface and the users' interface. In addition, many larger systems are structured so that providing a simple interface for the majority of users and a more extensive interface for expert users is a good idea. The expert users' interfaces (''complete interfaces'') tend to *#include* many more features than the average user would ever want to know about. In fact, the average users' interface can often be identified by eliminating features that require the inclusion of headers that define facilities that would be unknown to the average user. The term ''average user'' is not derogatory. In the fields in which I don't *have* to be an expert, I strongly prefer to be an average user. In that way, I minimize hassles.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

9.3.3 Include Guards [file.guards]

The idea of the multiple-header approach is to represent each logical module as a consistent, self-contained unit. Viewed from the program as a whole, many of the declarations needed to make each logical module complete are redundant. For larger programs, such redundancy can lead to errors, as a header containing class definitions or inline functions gets *#include*d twice in the same compilation unit (§9.2.3).

We have two choices. We can

[1] reorganize our program to remove the redundancy, or

[2] find a way to allow repeated inclusion of headers.

The first approach – which led to the final version of the calculator – is tedious and impractical for realistically-sized programs. We also need that redundancy to make the individual parts of the program comprehensible in isolation.

The benefits of an analysis of redundant *#include*s and the resulting simplifications of the program can be significant both from a logical point of view and by reducing compile times. However, it can rarely be complete, so some method of allowing redundant *#include*s must be applied. Preferably, it must be applied systematically, since there is no way of knowing how thorough an analysis a user will find worthwhile.

The traditional solution is to insert *include guards* in headers. For example:

```
// error.h:

#ifndef CALC_ERROR_H
#define CALC_ERROR_H

namespace Error {
	// ...
}

#endif		// CALC_ERROR_H
```

The contents of the file between the *#ifndef* and *#endif* are ignored by the compiler if *CALC_ERROR_H* is defined. Thus, the first time *error.h* is seen during a compilation, its contents are read and *CALC_ERROR_H* is given a value. Should the compiler be presented with *error.h* again during the compilation, the contents are ignored. This is a piece of macro hackery, but it works and it is pervasive in the C and C++ worlds. The standard headers all have include guards.

Header files are included in essentially arbitrary contexts, and there is no namespace protection against macro name clashes. Consequently, I choose rather long and ugly names as my include guards.

Once people get used to headers and include guards, they tend to include *lots* of headers directly and indirectly. Even with C++ implementations that optimize the processing of headers, this can be undesirable. It can cause unnecessarily long compile time, and it can bring *lots* of declarations and macros into scope. The latter might affect the meaning of the program in unpredictable and adverse ways. Headers should be included only when necessary.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

9.4 Programs [file.programs]

A program is a collection of separately compiled units combined by a linker. Every function, object, type, etc., used in this collection must have a unique definition (§4.9, §9.2.3). The program must contain exactly one function called *main*() (§3.2). The main computation performed by the program starts with the invocation of *main*() and ends with a return from *main*(). The *int* returned by *main*() is passed to whatever system invoked *main*() as the result of the program.

This simple story must be elaborated on for programs that contain global variables (§10.4.9) or that throw an uncaught exception (§14.7).

9.4.1 Initialization of Nonlocal Variables [file.nonlocal]

In principle, a variable defined outside any function (that is, global, namespace, and class *static* variables) is initialized before *main*() is invoked. Such nonlocal variables in a translation unit are initialized in their declaration order (§10.4.9). If such a variable has no explicit initializer, it is by default initialized to the default for its type (§10.4.2). The default initializer value for built-in types and enumerations is *0*. For example:

```
double x = 2;          // nonlocal variables
double y;
double sqx = sqrt(x+y);
```

Here, *x* and *y* are initialized before *sqx*, so *sqrt*(*2*) is called.

There is no guaranteed order of initialization of global variables in different translation units. Consequently, it is unwise to create order dependencies between initializers of global variables in different compilation units. In addition, it is not possible to catch an exception thrown by the initializer of a global variable (§14.7). It is generally best to minimize the use of global variables and in particular to limit the use of global variables requiring complicated initialization.

Several techniques exist for enforcing an order of initialization of global variables in different translation units. However, none are both portable and efficient. In particular, dynamically linked libraries do not coexist happily with global variables that have complicated dependencies.

Often, a function returning a reference is a good alternative to a global variable. For example:

```
int& use_count()
{
    static int uc = 0;
    return uc;
}
```

A call *use_count*() now acts as a global variable except that it is initialized at its first use (§5.5). For example:

```
void f()
{
    cout << ++use_count();    // read and increment
    // ...
}
```

The initialization of nonlocal static variables is controlled by whatever mechanism an

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

implementation uses to start up a C++ program. This mechanism is guaranteed to work properly only if *main*() is executed. Consequently, one should avoid nonlocal variables that require run-time initialization in C++ code intended for execution as a fragment of a non-C++ program.

Note that variables initialized by constant expressions (§C.5) cannot depend on the value of objects from other translation units and do not require run-time initialization. Such variables are therefore safe to use in all cases.

9.4.1.1 Program Termination [file.termination]

A program can terminate in several ways:
- By returning from *main*()
- By calling *exit*()
- By calling *abort*()
- By throwing an uncaught exception

In addition, there are a variety of ill-behaved and implementation-dependent ways of making a program crash.

If a program is terminated using the standard library function *exit*(), the destructors for constructed static objects are called (§10.4.9, §10.2.4). However, if the program is terminated using the standard library function *abort*(), they are not. Note that this implies that *exit*() does not terminate a program immediately. Calling *exit*() in a destructor may cause an infinite recursion. The type of *exit*() is

```
void exit(int);
```

Like the return value of *main*() (§3.2), *exit*()'s argument is returned to ''the system'' as the value of the program. Zero indicates successful completion.

Calling *exit*() means that the local variables of the calling function and its callers will not have their destructors invoked. Throwing an exception and catching it ensures that local objects are properly destroyed (§14.4.7). Also, a call of *exit*() terminates the program without giving the caller of the function that called *exit*() a chance to deal with the problem. It is therefore often best to leave a context by throwing an exception and letting a handler decide what to do next.

The C (and C++) standard library function *atexit*() offers the possibility to have code executed at program termination. For example:

```
void my_cleanup();

void somewhere()
{
        if (atexit(&my_cleanup)==0) {
                // my_cleanup will be called at normal termination
        }
        else {
                // oops: too many atexit functions
        }
}
```

This strongly resembles the automatic invocation of destructors for global variables at program termination (§10.4.9, §10.2.4). Note that an argument to *atexit*() cannot take arguments or return a

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

result. Also, there is an implementation-defined limit to the number of atexit functions; ***atexit*()** indicates when that limit is reached by returning a nonzero value. These limitations make ***atexit*()** less useful than it appears at first glance.

The destructor of an object created before a call of ***atexit*(*f*)** will be invoked after *f* is invoked. The destructor of an object created after a call of ***atexit*(*f*)** will be invoked before *f* is invoked.

The ***exit*()**, ***abort*()**, and ***atexit*()** functions are declared in **<*cstdlib*>**.

9.5 Advice [file.advice]

[1] Use header files to represent interfaces and to emphasize logical structure; §9.1, §9.3.2.
[2] *#include* a header in the source file that implements its functions; §9.3.1.
[3] Don't define global entities with the same name and similar-but-different meanings in different translation units; §9.2.
[4] Avoid non-inline function definitions in headers; §9.2.1.
[5] Use *#include* only at global scope and in namespaces; §9.2.1.
[6] *#include* only complete declarations; §9.2.1.
[7] Use include guards; §9.3.3.
[8] *#include* C headers in namespaces to avoid global names; §9.3.2.
[9] Make headers self-contained; §9.2.3.
[10] Distinguish between users' interfaces and implementers' interfaces; §9.3.2.
[11] Distinguish between average users' interfaces and expert users' interfaces; §9.3.2.
[12] Avoid nonlocal objects that require run-time initialization in code intended for use as part of non-C++ programs; §9.4.1.

9.6 Exercises [file.exercises]

1. (∗2) Find where the standard library headers are kept on your system. List their names. Are any nonstandard headers kept together with the standard ones? Can any nonstandard headers be *#include*d using the **<>** notation?
2. (∗2) Where are the headers for nonstandard library ''foundation'' libraries kept?
3. (∗2.5) Write a program that reads a source file and writes out the names of files *#include*d. Indent file names to show files *#included* by included files. Try this program on some real source files (to get an idea of the amount of information included).
4. (∗3) Modify the program from the previous exercise to print the number of comment lines, the number of non-comment lines, and the number of non-comment, whitespace-separated words for each file *#include*d.
5. (∗2.5) An external include guard is a construct that tests outside the file it is guarding and *include*s only once per compilation. Define such a construct, devise a way of testing it, and discuss its advantages and disadvantages compared to the include guards described in §9.3.3. Is there any significant run-time advantage to external include guards on your system.
6. (∗3) How is dynamic linking achieved on your system. What restrictions are placed on dynamically linked code? What requirements are placed on code for it to be dynamically linked?

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

7. (∗3) Open and read 100 files containing 1500 characters each. Open and read one file containing 150,000 characters. Hint: See example in §21.5.1. Is there a performance difference? What is the highest number of files that can be simultaneously open on your system? Consider these questions in relation to the use of ***#include*** files.
8. (∗2) Modify the desk calculator so that it can be invoked from ***main*()** or from other functions as a simple function call.
9. (∗2) Draw the ‘‘module dependency diagrams’’ (§9.3.2) for the version of the calculator that used ***error*()** instead of exceptions (§8.2.2).

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.